UNIVERSITY OF WATERLOO
DEPARTMENT OF COMPUTER SCIENCE

A Portable Assembler Writing Kit

Michael A. Malcolm
Gary R. Sager
Gary J. Stafford

November 1976
CS-76-48

UNIVERSITY OF WATERLOO
DEPARTMENT OF COMPUTER SCIENCE

A Portable Assembler Writing Kit

Michael A. Malcolm

Gary R. Sager

Gary J. Stafford

Department of Computer Science
University of Waterloo

November 1976

This research was supported by the National Research Council of Canada.

Preprint: To appear in the *IEEE Proceedings of the Conference on Mini and Micro Computers*, November 1976, Toronto.

A Portable Assembler Writing Kit

M. A. Malcolm
G. R. Sager
G. J. Stafford

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

abstract: The Last Assembler (TLA) is an "assembler kit" designed to support the portable programming system being developed at the University of Waterloo. TLA Assemblers are capable of cross or on-site operation .

Approximately 80% of the TLA code remains unchanged across all implementations. Another 10 to 15% of the code is generated automatically from descriptions of the target machine. The implementor must also supply a table of mnemonic opcodes, their values and "classes". The remaining 5 to 10% must be provided in the form of small well-specified functions which deal with error detection for each class of operand and the combining of operands with opcodes.

To date TLA has been used to generate assemblers for six machines. The average time to completion for each of these assemblers has been less than eight man-hours.

1. Introduction

In this paper we describe one component of a system being developed by the Portable System Software Project at the University of Waterloo. This project is developing software tools that can be used for programming on and for a variety of different machines. We have found, through using these tools for programming minicomputers in our own laboratory [1], that the effort required to convert the system for a new minicomputer is far less than the effort required to implement it initially. Furthermore, our portable system obviates the need to learn and maintain new vendor systems (when available!) thus making it easier to integrate a new machine into the laboratory. As the first step in this project, we have defined and implemented the language Eh [2], which is designed to facilitate portable systems programming. The Eh compiler is written in Eh and constructed to be easily converted to generate code for new machines. The output of Eh programs is relocated and linked with precompiled and assembled routines by ULD, a portable linking loader [3].

It is not possible to avoid assembly coding altogether, even though Eh provides the ability to emit machine instructions inline. We find that assembly language is most appropriate for filling interrupt vectors, for dispatching interrupt routines and for certain functions critical to system performance. Since our system will undoubtedly be incompatible with the assembler and loader supplied by the vendor, we must implement our own assembler to interface with Eh and ULD. But if we must write a complete assembler for each machine, the portability of our system is reduced. We have, therefore, implemented an "assembler-writing kit" which enables us to build an assembler for a new machine with a minimum of effort. Since we do very little assembly coding, the assemblers we build need not be 'efficient' or 'powerful'.

Before proceeding, we must define a few terms. We say that a program is **portable** over a set of environments if it is significantly easier to move to and maintain in those environments than to implement and maintain separately in those environments. By the term **environment** we mean the combination of a machine and system software necessary to execute a program. The **target environment** is the environment for which the output of the program is intended, and the **host environment** is the environment in which the program executes. All programs have host environments, but only certain system programs such as assemblers, have target environments. A program is **machine dependent** if it requires some particular set of host machine hardware features. A program is **machine specific** if it requires some particular set of target machine hardware features.

Thus, portability is a matter of degree, and portability problems may take many forms. The Eh compilers and TLA assemblers are portable in the sense that they are easily adapted for a new target as well as being easily moved to a new host. In this paper, we will focus on the design features of TLA which make it easily adapted to a new target.

2. Structure of the Assembler

There are a number of commercially available cross-assemblers (usually written in FORTRAN) which are portable in the sense that they execute in a wide range of host environments to produce assembled output for their target environment. We of course want this form of portability, but we also want our assembler to be portable in the sense that it can easily be modified to assemble for a new target machine with little effort. A term sometimes used to denote this type of portability is **adaptability.** We have built our assembler to be machine independent, and parameterized those parts which must change from one target to another.

On close inspection, one may observe that large portions of a typical assembler are simply algorithms which can be expressed in terms independent of the target or host environments. These algorithms include symbol table managment, processing of assembler directives, scanning of input lines, expression evaluation, formatting the output listing, and creation of the load module. Our experience and that of others [4] has shown that these algorithms constitute approximately 80% of the code in an assembler.

If this invariant portion of the code is written in a machine independent fashion, it is only necessary to code the remaining 20% in order to obtain an assembler which will assemble for and potentially execute on another machine.

The syntax for TLA assembler languages is given in appendix II. The languages include directives, pseudo operations, classes of machine instruction mnemonics, identifiers and constants, plus a set of operand symbols. The input scanner classifies tokens into these categories. Strings of alphabetic and numeric characters which begin with an alphabetic character are compared with symbols in the symbol table to categorize them as directive, pseudo operation, mnemonic or identifier. If the symbol is a mnemonic, it is further categorized into one of the instruction classes specified by the implementor (this is described later). Strings beginning with a numeric character, $ or ' are categorized as constants. The scanner computes the value based on the first character of the constant: 0 indicates base 8, $ indicates base 16, ' indicates the base specified by the last .BASE directive, otherwise the base is 10.

The 20% of the code which is machine specific is related to the pseudo operations, machine operations and error checking. The structure of these functions is predetermined both by the manner in which data is provided to them by the higher level functions and by the information they must pass on to the lower level functions. Because of this structure, it it possible to construct them in skeletal form from parameters describing the target machine. The construction of these modules is done by an interactive program called HELPER (see appendix III). The parameters of interest to HELPER include bits per byte, bits per addressable cell, the number of bits in the largest address, etc. Based on this information, HELPER outputs the skeletal forms of the functions which must be augmented by hand. Typically, the amount of code added by the implementor is 5 to 10% of the total assembler, or up to 250 out of a total of 2500 lines of code. In addition to these functions, the implementor must prepare a table of machine operation mnemonics and their values, and categorize each operation into a "class" which indicates the number and position of operands within the instruction.

Of the time spent by the implementor, approximately 50% is to build the table of machine operation mnemonics, classes and values, 5% for interaction with the HELPER and 45% to augment the code skeletons for directives and machine operations.

3. Expressions and Relocatability

Expression evaluation in TLA follows a simple set of rules: evaluation is left to right, but can be over-ridden with parentheses. The standard set of operations available include addition (+), subtraction (−), bit-wise *and* (&) and bit-wise *or* (|). The special symbols ++, −−, @, !, #, ↑, [and] are not processed by the expression evaluator, but their presence at either the beginning or ending of an expression is reported to the machine specific functions, which may interpret them as appropriate to the target machine. For example, the implementor may choose to have the symbol @ indicate indirection.

There are two basic problems to confront in expression evaluation: we must be able to complete the evaluation for the target machine independently of the precision and type of arithmetic available on the host. We have solved the problem of precision by storing all symbol values as strings of bytes and by performing all evaluations serially by bytes. This is one of the few places in TLA assemblers where we have found it necessary to make a significant sacrifice in time and space efficiency to gain in portability. The problem of matching the type of arithmetic on the target is avoided by standardizing on two's complement. This is one of the most common forms of data representation in use; in cases where the target is not a two's complement machine, the implementor must add code to convert the two's complement representation to that of the target.

A major problem which must be resolved in the machine invariant code of the assembler is the question of evaluating "relocatable" expressions which are subject to modification at load time. Some expressions can only be evaluated by the loader. Here we have the advantage of knowing fully the operational characteristics and capabilities of our relocating loader, ULD [3]. Basically, our loader allows programs to be loaded in up to 8 separate relocatable "sections" and have up to 8 types of field it can relocate. The user indicates the relocatable sections of his program with the directive:

.rel <expr>

where <expr> must evaluate to a number between 0 and 7. The value must be chosen to be compatible with the relocations used by the Eh compiler, so the user is required to have some knowledge of the implementation of Eh before doing any serious assembly language programming.

Values in the TLA symbol table have an associated attribute which may be either absolute (i.e. known at assembly time) or relocatable. In the case of relocatable values, a further distinction is made to indicate the relocation which will apply at load time. External symbols have attributes which cannot be determined until load time, so TLA assemblers give them a relocatable attribute which will not match that of any other symbol at assembly time.

As an expression is evaluated, each operation and the attributes of its operands are used to determine the attribute of the result. The rule for deriving attributes of results is best presented in the form of a table:

		right operand		
	+	A	R	R′
left operand	A	A	R	R′
	R	R	E	E
	R′	R′	E	E

In this table, A stands for absolute, R is a relocatable and R′ is a relocatable whose attribute is different from R. The symbol E is used to indicate an error. The table then tells us, for example, that it is not possible at any point during an expression evaluation to add a relocatable to another relocatable. Tables for the other operators are:

right operand

−	A	R	R′
A	A	E	E
R	R	A	E
R′	R′	E	A

left operand

right operand

&	A	R	R′
A	A	E	E
R	E	E	E
R′	E	E	E

left operand

right operand

|	A	R	R′
A	A	E	E
R	E	E	E
R′	E	E	E

left operand

4. Some Implementation Details

The interactive session between the implementor and the HELPER program, as illustrated in appendix III, results in the creation of three files. These files contain manifests (a weak form of textual macro) which define machine-specific parameters in the invariant code, externals containing byte-serial representations of constants required by the implementor, and skeletal forms of functions to implement the .dc1, .dc2, .dc3 and .dc4 pseudo operations (these are used to assemble expression values into memory locations). Since the HELPER program generates these completely for most machines, we will not discuss them here.

The implementor must provide opcode functions for each class of operation code he has defined in the opcode table. These functions must perform a number of tasks, as outlined below:

```
OPCOn()
  {
    declare externals
    declare local variables

    if (pass 1)
        advance to end of input line

    if (pass 2)
      {
        call expression evaluator for each
            operand
        verify that each operand is in
            range and has the proper
            relocation attribute
        output assembled bytes to listing
        output assembled bytes and reloca-
            tion data to load module
        copy remainder of line to listing
      }

    increment location counter
  }
```

For some machines (e.g. the PDP11), the evaluation of the operands may reveal that the location counter is to be incremented more than once. In such a case, the OPCO function must call the scanner to check for special symbols during pass 1.

Since the operations which must be performed in a typical opcode function are well standardized, lower level functions are provided to aid the implementor in accomplishing each of the tasks in the outline above. The use of byte string representations and byte-serial arithmetic complicates the task somewhat, but these low level functions do range checking, output of listings in octal or hexadecimal, and other common duties. Thus, the coding of the opcode functions is largely a matter of selecting the correct arguments to the lower level functions.

Finally, for the output of the relocatable load module, there are two functions which add loading information. In order to interface the output modules correctly with modules output by the Eh compiler, the implementor will have to obtain information regarding the types of fields which can be relocated (the person who is writing the compiler will know this). If these are not adequate, it should be possible to define more fields for use by TLA.

5. Conclusion

We have used TLA to generate assemblers for the Honeywell H6060, Texas Instruments TI-990, Data General NOVA, Microdata 1600/30, Interdata 70 and Motorola M6800. The average time to completion for each of these assemblers is less than eight man-hours for a person experienced with TLA but not with the target machine. Time to completion for the Microdata by a person unfamiliar with both TLA and the target machine was approximately 13 hours.

The H6060, TI-990 and NOVA assemblers have been used to write support software for Thoth, our portable real-time executive [5].

A more formal approach to solving some of the problems in automating the generation of assemblers has been published by Wick [6]. It is difficult to compare the effectiveness of Wick's approach with that discussed here.

BIBLIOGRAPHY

[1] Malcolm, M. A. and G. R. Sager "Report on the Real-Time/Minicomputer Laboratory", Technical Report CS-76-11, Feb 1976, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

[2] Braga, R. S. C. "Eh Reference Manual", Technical Report CS-76-45, Oct 1976, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

[3] Braga, R. S. C. , M. A. Malcolm and G. R. Sager "A Portable Linking Loader", Symposium on Trends and Applications 1976: MICRO and MINI Systems, May 1976, pp 124-128.

[4] Mueller, R. A. "Automatic Generation of Microcomputer Software", Master's Thesis, Apr 1976, Dept. of Mechanical Engineering, Colorado State Universtiy, Fort Collins, CO.

[5] Melen, L. S. "A Portable Real-Time Executive, Thoth", Master's Thesis, Oct 1976, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

[6] Wick, J. D. "Automatic Generation of Assemblers", Ph. D. Thesis, Dec 1975, Dept. of Computer Science, Yale University.

Appendix I: TLA Directives and Pseudo Operations

The following directives are used in all implementations of TLA.

1. .ALIGN : used to align the next instruction on an addressable cell which is a multiple of the expression (which must be a power of two).

2. .BASE : specifies the base to be used in evaluating constants which start with the escape character '.

3. .DC1, .DC2, .DC3, .DC4 : define a constant Eh-word having 1, 2, 3 or 4 sub-fields.

4. .DS : reserves amount of storage indicated by expression.

5. .ENT : specifies identifiers which will be available to other programs.

6. .EQU : equates the value and attribute of an identifier to the value and attribute of the expression.

7. .EXT : specifies an identifier which is defined in another program.

8. .LOC : sets the value and attribute of the location counter to that of the expression.

9. .PAGE : advances listing to new page.

10. .REL : sets the attribute of the location counter to the value of the expression.

11. .TITL : specifies module name for use by library editor and loader.

Appendix II: Syntax of TLA Source Programs

The following characters are accepted by the TLA input scanner:

1. alphabetics: A B C D E F G H I J K L M N O P Q R T U V W X Y Z . _. All alphabetics are translated to upper case upon input to the scanner.
2. numerics: 0 1 2 3 4 5 6 7 8 9
3. delimiters: + − ( ) [ ] ! ' $ | & : ; , \ # @ *n and blank.

Several characters deserve special attention: *n is a "newline" used to delimit input lines; \ and ; will cause the scanner to ignore the remainder of the input line (this is for entering comments).

In addition to the delimiter characters, the tokens recognized by the scanner are:

1. special tokens: ++, −−
2. directives: .titl, .align, .base, .page, .ext, .ent, .equ, .loc, .rel, .ds, .equ, .end
3. pseudo operations: .dc1, .dc2, .dc3, .dc4
4. implementor-defined opcode mnemonics
5. user-defined constants and identifiers

It is not possible to give a syntax which applies fully to all assemblers implemented using TLA. In particular, the exact form of the argument list will vary from machine to machine, and interpretations of *special* symbols will vary. The following grammar will give the flavor of TLA grammars.

program	::=	*line−list* .end
line−list	::=	*line line−list*
line	::=	*label−list statement* '*n'
label−list	::=	*label label−list* null
label	::=	identifier :

statement ::=
.align *expr*
.base *expr*
.ds *expr*
.ent *ident−list*
.ext *ident−list*
.loc *expr*
.page
.rel *expr*
.titl identifier
identifier .equ *expr*
.dc1 *expr*
.dc2 *expr* , *expr*
.dc3 *expr* , *expr* , *expr*
.dc4 *expr* , *expr* , *expr* , *expr*
mnemonic *argument−list*
null
ident−list , identifier

argument−list ::=
operand
argument−list , *operand*

operand ::=
expr
expr [*expr*]

special ::=
++
−−
@
!
#
↑

expr ::=
(*expr*)
expr & *expr*
expr + *expr*
expr − *expr*
expr | *expr*
− *expr*
special expr
expr special
identifier
constant

Appendix III: An Example Session with the Helper

I am here to help get your new assembler started.
Have you used me before (yes or no) ? YES

What machine are you writing an assembler for? INTERDATA 70
What is the catalog name for this assembler? PORT/TLA/INTER

Now tell me about the INTERDATA 70.
  How many bits in a byte? 8
  How many bits in the smallest addressable cell? 8
  How many bits in the largest machine address? 16
  How many bits in the largest opcode? 16
  How many bits in the smallest instruction? 16
  How many bytes does a no-op instruction take? 2
  What is the numeric value of the no-op instruction? $0800

Now tell me about how Eh is implemented on the INTERDATA 70.
  How many bits are there in an Eh-word? 16
  How many relocation bases are used? 1

What is the maximum number you will allow in a .ds? 32768

Will you need a .dc1? YES
  What is the maximum value allowed? 32767
  What is the minimum value allowed? -32768
  Is the operand relocatable? YES
  Which relocation descriptor? 0

Will you need a .dc2? NO

Will you need a .dc3? NO

Will you need a .dc4? NO

Do you have type 1 opcodes? YES
  Do you need to do range checking for operands? YES
    Enter the minimum and maximum. 0 15
    Another pair?

Do you have type 2 opcodes? YES
  Do you need to do range checking for operands? YES
    Enter the minimum and maximum. 0 15
    Another pair?

```
Do you have type 3 opcodes? YES
   Do you need to do range checking for operands? YES
      Enter the minimum and maximum. 0 15
      Another pair? -32768 65536
      Another pair?

Do you have type 4 opcodes? YES
   Do you need to do range checking for operands? YES
      Enter the minimum and maximum. 0 15
      Another pair? -32678 65536
      Another pair?

Do you have type 5 opcodes? NO

The INTERDATA 70 assembler has had three files generated for it.
PORT/TLA/INTER/MANIFS contains manifests.
PORT/TLA/INTER/EXTERN contains externals.
PORT/TLA/INTER/GEN contains functions.

Best of luck with the rest of the assembler!
```